

Romberg Quadrature

Mth 351 Winter 1999

Bent E. Petersen

Filename: romberg_quadrature.mws

This worksheet illustrates how Romberg quadrature is carried out by extrapolation from the trapezoidal rule.

Maple's student package contains procedures to carry out trapezoidal and Simpson's quadrature. Rather than using these procedures we will define our own since it is not difficult to do so, and it is instructive. The procedures in the student package are designed to work with expressions (in some variable). We will design ours to work with functions.

```
> restart;
```

First here is the trapezoidal rule T

```
> T:=proc(f,r::range,n)
>   local k,s,h,a,b;
>   if type(n, numeric) then
>     if not type(n, posint) then ERROR("n must be a positive
integer"); fi;
>   fi;
>   a:=op(1,r); b:=op(2,r);
>   h:=(b-a)/n;
>   s:=f(a)+f(b)+2*sum(f(a+k*h),k=1..n-1);
>   s*h/2;
> end;
```

Note we check n carefully if it is numeric, but we do not check it if it is symbolic. The parameter f should be a function, of course, but we do not check it either, because we want to be able to do symbolic manipulation with unspecified symbols.

Here's our Simpson's rule procedure. Note n must be a positive even integer or symbolic.

```
> S:=proc(f,r::range,n)
>   local k,m,s,h,a,b;
>   if type(n, numeric) then
>     if not type(n, posint) then ERROR("n must be a positive
integer"); fi;
>     if irem(n,2)=1 then ERROR("n must be an even integer"); fi;
>   fi;
```

```

> m:=n/2; a:=op(1,r); b:=op(2,r);
> h:=(b-a)/n;
> s:=f(a)+f(b)+4*sum(f(a+(2*k+1)*h),k=0..m-1)+2*sum(f(a+2*k*h),k=1..
  m-1);
> s*h/3;
> end:

```

The error in the trapezoid rule can be expanded in an asymptotic series in positive even powers of the step size $h = \frac{b-a}{n}$. In Richardson extrapolation we use two different step sizes (usually h and $2h$) to eliminate the top order term in an error expansion. In the case of the trapezoidal rule, the result is an order 4 rule which turns out to be Simpson's rule. If we extrapolate again we obtain an order 6 rule, and so on. If we start with the trapezoidal rule with 2^m subintervals and extrapolate m times we obtain the m^{th} Romberg estimate of the integral. The calculation of course involves the trapezoidal rules with 2^k subintervals, $k = 0, 1, \dots, m$.

Here's the m^{th} Romberg estimate:

```

> R:=proc(f,r::range,m)
> local k,j,R;
> if type(m, numeric) then
>   if not type(m,nonnegint) then ERROR("m must be a non-negative
  integer"); fi; fi;
> for k from 0 to m do
>   R[0,k]:=T(f,r,2^k); od;
> for j from 1 to m do
>   for k from j to m do
>     R[j,k]:=1/(4^j-1)*(4^j*R[j-1,k]-R[j-1,k-1]);
>   od; od;
> R[m,m];
> end:

```

Note the 0^{th} estimate is the trapezoidal rule with one interval (as it should be)

```

> R(f,a..b,0);

```

$$\frac{1}{2}(f(a) + f(b))(b - a)$$

Note if we had typed f as a function in the trapezoidal code above we would not be able to obtain this simple expression without actually supplying a *function* for f .

The 1^{st} estimate is Simpson's rule with two subintervals, again as expected

```

> simplify(R(f,a..b,1)-S(f,a..b,2));

```

Our confidence bolstered by these two checks let's try some comparisons.

The code for $R(f, a..b, m)$ is particularly susceptible to loss of significance error (due to roundoff). Maple computes $R(f, a..b, m)$ exactly symbolically, but a sum of very many terms of roughly the same magnitude (but varying signs) results. Converting the large sum that results to floating point turns out to be quite a challenge for `evalf()`.

In the examples below the relative error is first computed symbolically. Then it is converted to floating point. Thus it is possible to control the roundoff error in computing the error simply by increasing the precision of conversion. With high enough precision we see only the truncation error.

Example 1

```
> rng1:=0..Pi: ex1:=Int(sin(x),x=rng1);
```

$$ex1 := \int_0^{\pi} \sin(x) dx$$

```
> `trapezoidal 32`; ex:=T(sin,rng1,32):evalf(ex); rel_error =
evalf((ex1-ex)/ex1,10); rel_error = evalf((ex1-ex)/ex1,24);
```

trapezoidal 32

1.998393361

rel_error = .0008033195000

rel_error = .000803319514927706701560000

```
> `Simpson's 32`; ex:=S(sin,rng1,32):evalf(ex); rel_error =
evalf((ex1-ex)/ex1,10); rel_error = evalf((ex1-ex)/ex1,24);
```

Simpson's 32

2.000001034

rel_error = -.5170000000 10⁻⁶

rel_error = -.516684706500345535000000 10⁻⁶

```
> `Romberg 5 (32)`; ex:=R(sin,rng1,5):evalf(ex,10); rel_error =
evalf((ex1-ex)/ex1,10); rel_error = evalf((ex1-ex)/ex1,24);
rel_error = evalf((ex1-ex)/ex1,36);
```

Romberg 5 (32)

1.999999999

rel_error = .7326650000 10⁻¹⁰

rel_error = -.660522072880558990000000 10⁻¹²

rel_error = -.660522072878470630759082967130000000 10⁻¹²

Only the Romberg estimates showed much sensitivity to possible roundoff here. On the other hand is far more accurate.

Example 2

```
> rng2:=8..12: ex2:=Int(exp(x),x=rng2);
```

$$ex2 := \int_8^{12} e^x dx$$

```
> `trapezoidal 32`; ex:=T(exp,rng2,32):evalf(ex); rel_error =  
evalf((ex2-ex)/ex2,10); rel_error=evalf((ex2-ex)/ex2,24);
```

trapezoidal 32

159981.8181

rel_error = -.001301744031

rel_error = -.00130174437523008390309158

```
> `Simpson's 32`; ex:=S(exp,rng2,32):evalf(ex); rel_error =  
evalf((ex2-ex)/ex2,10); rel_error = evalf((ex2-ex)/ex2,24);
```

Simpson's 32

159774.0497

rel_error = -.1353945107 10⁻⁵

rel_error = -.135381799023985119750504 10⁻⁵

```
> `Romberg 5 (32)`; ex:=R(exp,rng2,5):evalf(ex); rel_error =  
evalf((ex2-ex)/ex2,10); rel_error = evalf((ex2-ex)/ex2,24);  
rel_error = evalf((ex2-ex)/ex2,40);
```

Romberg 5 (32)

159773.8333

rel_error = .8199089752 10⁻⁹

rel_error = -.516058974550545043734913 10⁻¹¹

rel_error = -.5160589745493993176389342284719015293670 10⁻¹¹

The sensitivity to precision in the Romberg quadrature shows up clearly here. Lets try to using an unrealistically large number of nodes:

```
> `trapezoidal 1024`; ex:=T(exp,rng2,1024):evalf(ex); rel_error =  
evalf((ex2-ex)/ex2,10); rel_error=evalf((ex2-ex)/ex2,24);
```

trapezoidal 1024

159774.0504

rel_error = -.1358683055 10⁻⁵

rel_error = -.127156543183255675280761 10⁻⁵

```

> `Simpson's 1024`; ex:=S(exp,rng2,1024):evalf(ex); rel_error =
evalf((ex2-ex)/ex2,10); rel_error = evalf((ex2-ex)/ex2,24);
      Simpson's 1024
      159773.8330
      rel_error = .3417330538 10-9
      rel_error = -.129350122621426610125841 10-11
> st:=time():`Romberg 10 (1024)`; ex:=R(exp,rng2,10):evalf(ex);
rel_error = evalf((ex2-ex)/ex2,10); rel_error =
evalf((ex2-ex)/ex2,24); rel_error = evalf((ex2-ex)/ex2,40); `cpu
time` = time()-st;
      Romberg 10 (1024)
      159773.8535
      rel_error = -.1277493915 10-6
      rel_error = .951770366483422762384591 10-22
      rel_error = .2127231929655942044271156335879740668310 10-37
      cpu time = 5.530

```

When relative error changes so much with precision it is almost certain that we are seeing mostly roundoff error. The Romberg quadrature is extremely accurate, but one has to use high precision to achieve the accuracy. In any event Romberg is the clear winner here as far truncation error is concerned but more work is required to realize the advantages of the Romberg method.

From a philosophical point of view the difficulty with loss of significance should be expected. Extrapolation depends upon cancellation and so is bound to be susceptible to loss of significance from roundoff unless suitable precautions are taken. For example, if the major part of the cancellation occurs symbolically rather than in the context of floating point numbers, we would reduce the problem.

A small possible improvement is to re-arrange the iterative part of the Romberg calculation so that $R[j,k]$ is obtained by adding a fairly small number to $R[j-1,k]$ (at least for large j). It turns out this idea has no effect here, probably because Maple carries out the calculation symbolically and ultimately obtains the same symbolic expression as before. Here's a demonstration:

```

> Rb:=proc(f,r::range,m)
> local k,j,R;
> if type(m, numeric) then
>   if not type(m,nonnegint) then ERROR("m must be a non-negative
integer"); fi; fi;
> for k from 0 to m do
>   R[0,k]:=T(f,r,2^k); od;
> for j from 1 to m do
>   for k from j to m do

```

```

> R[j,k]:=R[j-1,k]+(R[j-1,k]-R[j-1,k-1])/(4^j-1);
> od; od;
> R[m,m];
> end:

```

Example 3

Let's test Rb to verify there is no improvement:

```

> rng3:=8..12: ex3:=Int(exp(x),x=rng3);

```

$$ex3 := \int_8^{12} e^x dx$$

```

> st:=time():`Romberg b 10 (1024)`; ex:=Rb(exp,rng3,10):evalf(ex);
rel_error = evalf((ex3-ex)/ex3,10); rel_error =
evalf((ex3-ex)/ex3,24); rel_error = evalf((ex3-ex)/ex3,40); `cpu
time`=time()-st;

```

Romberg b 10 (1024)

159773.8535

rel_error = -.1277493915 10⁻⁶

rel_error = .951770366483422762384591 10⁻²²

rel_error = .2127231929655942044271156335879740668310 10⁻³⁷

cpu time = 6.296

We can try to improve efficiency a bit by noting that the trapezoidal rule with $2m$ subintervals can be calculated from the trapezoidal rule with m subintervals by doing only an additional m function evaluations. Here is the code which produces the trapezoidal quadrature with n subintervals from the one with $\frac{n}{2}$ subintervals, n even.

```

> TA:=proc(f,r::range,n,T) # T should be n/2 trapezoidal result
> local k,h,a,b,m;
> if type(n, numeric) then
> if not type(n, posint) then ERROR("n must be a positive
integer"); fi;
> if irem(n,2)=1 then ERROR("n must be an even integer"); fi;
> fi;
> m:=n/2; a:=op(1,r); b:=op(2,r);
> h:=(b-a)/n;
> (1/2)*T+h*sum(f(a+(2*k-1)*h),k=1..m);
> end:

```

Let's introduce this code in Rb above:

```
> Rc:=proc(f,r::range,m)
> local k,j,R;
> if type(m, numeric) then
>   if not type(m,nonnegint) then ERROR("m must be a non-negative
integer"); fi; fi;
> R[0,0]:=T(f,r,1);
> for k from 1 to m do
>   R[0,k]:=TA(f,r,2^k,R[0,k-1]); od;
> for j from 1 to m do
>   for k from j to m do
>     R[j,k]:=R[j-1,k]+(R[j-1,k]-R[j-1,k-1])/(4^j-1);
>   od; od;
> R[m,m];
> end:
```

Example 4

```
> rng4:=8..12: ex4:=Int(exp(x),x=rng4);
```

$$ex4 := \int_8^{12} e^x dx$$

```
> st:=time():`Romberg c 10 (1024)`; ex:=Rc(exp,rng4,10):evalf(ex);
rel_error = evalf((ex4-ex)/ex4,10); rel_error =
evalf((ex4-ex)/ex4,24); rel_error = evalf((ex4-ex)/ex4,40);`cpu
time`=time()-st;
```

Romberg c 10 (1024)

159773.8334

rel_error = -.2072304288 10⁻⁸

rel_error = -.433675518147383823063553 10⁻²²

rel_error = -.4961200360367827031718678261371188420332 10⁻³⁷

cpu time = 10.046

Our quest for efficiency seems headed in the wrong direction! More detailed timing and a lot more information about how Maple works in the background is needed. I'll leave it for you to experiment with!

We could, for a given m , write the Romberg rule as a weighted sum of the evaluations of f at equispaced abscissas. This would give us a convenient formula. In applications, the Romberg rule is

often applied recursively, with the change between subsequent results used as an estimate of the error. This can be done quite efficiently, since previous calculations are recycled. Moreover, the error estimate is often good enough to be used as a stopping condition (by comparing it with a pre-assigned desired precision). We lose these conveniences if we pass to a formula, but on the other hand the weights are positive, and loss of precision is not a problem (for positive integrands), or is manageable (for integrands that change sign).

The weights can be computed by a simple technique. Just have Maple compute the Romberg estimates of the integrals of suitable interpolation polynomials. The polynomials you use are those that are 1 at one node and 0 at the others.

Let's look at the case $m = 2$. Here we have $2^m + 1 = 5$ nodes. Let's take $a = 0$ and $b = 1$.

```
> xc:=[0,1/4,1/2,3/4,1]: yc[0]:=[1,0,0,0,0]: yc[1]:=[0,1,0,0,0]:
  yc[2]:=[0,0,1,0,0]: yc[3]:=[0,0,0,1,0]: yc[4]:=[0,0,0,0,1]:
> for k from 0 to 4 do pc[k]:=unapply(interp(xc,yc[k],x),x); od;
```

$$pc_0 := x \rightarrow \frac{32}{3}x^4 - \frac{80}{3}x^3 + \frac{70}{3}x^2 - \frac{25}{3}x + 1$$

$$pc_1 := x \rightarrow -\frac{128}{3}x^4 + 96x^3 - \frac{208}{3}x^2 + 16x$$

$$pc_2 := x \rightarrow 64x^4 - 128x^3 + 76x^2 - 12x$$

$$pc_3 := x \rightarrow -\frac{128}{3}x^4 + \frac{224}{3}x^3 - \frac{112}{3}x^2 + \frac{16}{3}x$$

$$pc_4 := x \rightarrow \frac{32}{3}x^4 - 16x^3 + \frac{22}{3}x^2 - x$$

```
> for k from 0 to 4 do r[k]:=R(pc[k],0..1,2); od;
```

$$r_0 := \frac{7}{90}$$

$$r_1 := \frac{16}{45}$$

$$r_2 := \frac{2}{15}$$

$$r_3 := \frac{16}{45}$$

$$r_4 := \frac{7}{90}$$

These weights are well-known! They correspond to Boole's rule.

```
> RS4:=f->sum(r[k]*f(k/4),k=0..4); k:=evaln(k):
```

$$RS4 := f \rightarrow \sum_{k=0}^4 r_k f\left(\frac{1}{4}k\right)$$

```
> RS4(x->sin(Pi*x)); evalf(%);
```

$$\frac{2}{15} + \frac{16}{45}\sqrt{2}$$

```
.6361648221
```

```
> int(sin(Pi*x),x=0..1); evalf(%);
```

$$2 \frac{1}{\pi}$$

```
.6366197722
```

Note, for the interval $[0,1]$ the weights will be rational numbers, so in principle we can compute them exactly. In practice, of course, there is the risk that the numerators and denominators may become inconveniently large,

Experiment!